5

BUILDING A CHIP-8 VIRTUAL MACHINE

In this chapter, we're going to develop a version of a virtual machine known as CHIP-8, a platform from the early days of personal computing that was primarily used for playing games. Although our program will be able to play CHIP-8 games, it's not the games themselves that interest us—it's what building a CHIP-8 virtual machine can teach us about low-level programming and how a computer works at the register and instruction levels. These insights make building a CHIP-8 virtual machine a popular first step into the world of programming emulators.

Virtual Machines

Think of a *virtual machine (VM)* as a computer that's defined wholly in software. Programs that are designed to run in a VM can run on any platform that has an implementation of that VM. In this way, VMs enable truly portable software.

VMs are closely related to emulators. An *emulator* is a piece of software that's pretending to be a piece of hardware. This enables programs that were written for that hardware to run on other machines that lack the hardware. An emulator must follow the specification for the original hardware carefully so that it re-creates all the functionality that the unknowing programs running on the emulator expect. I say *unknowing* because the software running on an emulator has no idea it isn't running on the real hardware; the emulator had better work exactly like the original hardware if the program is going to function correctly.

A VM is also a piece of software that closely follows a specification of an environment that software runs on top of. The difference is that while an emulator follows a hardware specification, a VM follows a specification that may be wholly defined as an abstraction in software terms.

Although one is a hardware specification and one is a software specification, implementing a simple emulator is quite similar to implementing a simple VM. In fact, they're so similar that while the project completed in this chapter is technically a VM project, it's very commonly suggested as a first emulation project. If you're a newcomer to the emulator development community asking where you should start, CHIP-8 is almost always the answer.

Perhaps the most famous VM is the Java Virtual Machine (JVM). When Java first came out in the mid-1990s, its "write once, run anywhere" philosophy was touted. JVMs were developed for all major operating systems (Windows, Linux, Mac OS, and so on), and the same Java program could be compiled into the JVM's native bytecode format and run on any computer with a JVM unchanged, regardless of the underlying platform. That's still true today, but Java's original write-once-run-anywhere niche has largely been supplanted by web applications.

The CHIP-8 VM comes from a much earlier era. In the 1970s, Joseph Weisbecker was a pioneering engineer who developed one of the first 8-bit microprocessors, the RCA 1802. He and RCA built an early personal computer using his invention.[1] He wanted to have a way to program games for the machine in a higher-level language than machine code, so he developed CHIP-8 (and its accompanying opcode language). His daughter, Joyce Weisbecker, would go on to use CHIP-8 to become the first published female video game developer.[2] In the 1980s, CHIP-8 was ported to many other platforms, including many graphing calculators. It therefore became a truly portable VM, analogous to an early form of how we think about VMs today.

The CHIP-8 Virtual Machine

The CHIP-8 VM was originally designed for the incredibly resource-constrained personal computers of the late 1970s, like the COSMAC VIP. Released in 1977, the COSMAC VIP had an RCA 1802 8-bit microprocessor running at less than 2 megahertz (MHz), 2KB of RAM (expandable to 4KB), and a 512-byte ROM. It also had specialized chips for displaying 1-bit graphics at a resolution of up to 64×128, reading and writing cassette tapes, and playing a beep.[3]

It's amazing by today's standards that anything of value could have been programmed on a machine like the COSMAC VIP, yet it was designed for video games. In fact, those games even ran through another layer of abstraction, the CHIP-8 VM. The most popular video game console of the era, the Atari 2600, was also released in 1977 and had specifications that were in the same ballpark. These limitations were simply par for the course.

When programming a VM or an emulator, the performance of the tools you're using is a paramount concern. The VM or emulator adds another layer of abstraction between the program and the hardware, and each layer of abstraction generally comes with some performance cost. To achieve the intended speed of the original system, overhead has to be kept to a minimum, and some programming languages (or rather, some programming languages' primary runtime implementations) get in the way. This is why it's common to see VMs and emulators programmed in low-level languages like C, C++, and Rust. That said, considering how limited CHIP-8's original target hardware was, it's not difficult to create a performant CHIP-8 VM today on any modern system. Even a relatively slow programming language runtime like CPython is sufficient. You wouldn't want to program a cutting-edge game console emulator in Python, or a JVM. But CHIP-8? Python is more than fine for that.

To understand CHIP-8, let's start by discussing its registers and memory layout. Then, I'll provide a general overview of the instructions that the VM can execute, before getting into the nitty-gritty details of an implementation.

Registers and Memory

On a physical microprocessor, *registers* are the absolute fastest memory available. They sit directly within the microprocessor and don't require the latency of accessing another chip. Putting data in registers is often the only way to manipulate it, since most data manipulation instructions (for example, arithmetic) that a microprocessor supports operate on data within the registers. Separate load/store instructions transfer data between the registers and external RAM.

When it comes to registers, there's a classic time-versus-space trade-off: the registers are the fastest storage locations to hold data, but they're extremely limited in size. For example, a typical 8-bit microprocessor of the late 1970s may only have had a few 8-bit registers (yes, each can only hold a single byte), but it could address dozens of kilobytes of external RAM.

Most VMs, like the CHIP-8, also have registers, but those registers don't always map directly to physical hardware registers on the microprocessor. As such, they're not necessarily any faster than RAM. That may seem odd, but the registers provide a substrate that the instructions can operate on. There's also nothing stopping a particular implementation of the VM from mapping the virtual registers to real hardware registers for a performance gain—as long as the number of virtual registers doesn't exceed the number of physical registers.

NOTE *In the following discussion, the same names are used to refer to the CHIP-8 registers as will be used in the Python code for the implementation.*

The CHIP-8 VM has 16 general-purpose 8-bit registers, referred to as `v[0]` through `v[15]`. They can be used for any kind of data, and all the main arithmetic and logic instructions operate on these registers. Of these general-purpose registers, `v[15]` (or `v[0xF]` in hexadecimal) is special in that it's used for holding a flag. The index register, `i`, is for manipulations across multiple memory locations at once and for indicating where data that needs to be drawn to the screen exists in memory. The program counter, `pc`, is a special register that keeps track of the memory address of the next instruction to be executed.

The vs, `i`, and `pc` constitute the main registers, but they're backed up by a couple pseudo-registers for timing. These two bytes, `delay_timer` and `sound_timer`, are used for implementing a pause in the game or indicating how long the sound of a beep should be played. There are specialized instructions for modifying these timers. All the registers are listed in Table 5-1. The registers were originally described in the RCA COSMAC VIP CDP18S711 Instruction Manual.[4]

Table 5-1: CHIP-8 Registers and Pseudo-Registers

Register	Name	Description
`v[0]` to `v[14]`	General-purpose registers	Each can hold any kind of 8-bit data.
`v[15]`	Flag register	Stores a flag (1 or 0) after certain operations, like a carry flag after addition.
`pc`	Program counter	Keeps track of the 16-bit address in memory of the current instruction being executed.
`i`	Memory index register	Stores a 16-bit address used for completing instructions that span multiple contiguous places in memory.
`delay_timer`	Delay timer	Stores an 8-bit value that's decremented 60 times per second until it reaches 0.
`sound_timer`	Sound timer	Stores an 8-bit value that's decremented 60 times per second until it reaches 0; while it's above 0, a beep is played by the computer speaker.

A typical CHIP-8 VM has 4KB of general-purpose RAM. This is in line with the COSMAC VIP when loaded with expansion memory. However, there's a catch: on the VIP, the first 512 bytes of memory had to contain the code for the actual CHIP-8 VM itself (yes, the whole VM fit into just 512 bytes of machine code—think about that as we write our version). That left only 3.5KB of usable RAM. To be backward compatible today, our VM must also reserve the first 512 bytes of RAM.

Instructions

The CHIP-8 VM was largely used to program games, so it includes specialized instructions for actions like moving sprites and playing a beep. Those sit alongside all the mundane, utilitarian instructions you'd find in any microprocessor instruction set or low-level programming language—instructions for manipulating memory, doing arithmetic, overseeing control flow, handling timers, and managing the display. In total, there are 35 instructions that we'll be implementing. All the instructions are specified in hexadecimal—see the "Hexadecimal" box for more on that numbering system.

HEXADECIMAL

Hexadecimal, or *base-16*, is the number system typically used for working with low-level bytes on computing systems (RAM addresses, CPU instructions, and the like). It can more compactly and consistently refer to values in bytes than binary or standard decimal (base-10, the number system we're used to). For instance, you can represent any 8-bit number using two hexadecimal digits, and helpfully, each of those two digits corresponds to exactly half of the byte when written out in binary (half of a byte is known as a *nibble*). If you were a programmer in the 1970s or 1980s, you would work with hexadecimal often, but today the average Python developer seldom uses it outside of low-level programming.

In hexadecimal, in addition to the 10 symbols 0–9, six further symbols are provided, A–F, corresponding to the decimal values 10–15. In Python, hexadecimal literals start with the `0x` prefix. For example, `0xFF` is the same as the decimal number `255`, or the binary number `0b11111111`. One F in the hexadecimal version refers to the first half of the ones in the binary version (1111), and the other F refers to the second set of ones (1111). This is the maximum value of 1 byte. To illustrate the conversion more clearly, the hexadecimal number `0xF0` can be written in binary as `0b11110000`, with the F for the 1111 and the 0 for the 0000.

To convert from hexadecimal to decimal, multiply each hex digit from right to left by a power of 16, starting with 16^0. For example, `0xFF` can be rewritten as $(15 \times 16^0) + (15 \times 16^1)$. The right digit (F) becomes $15 \times 1 = 15$, the left digit becomes $15 \times 16 = 240$, and $240 + 15 = 255$. Here's another example: `0xA5B` is $(11 \times 16^0) + (5 \times 16^1) + (10 \times 16^2)$. This is equivalent to 2,651 in decimal.

The instructions are here as a quick reference and to give you a sense of the "lay of the land." We'll get into the details of how each instruction works in the code, but the reality is that most of the code is pretty self-explanatory based on the instruction descriptions. The vast majority of instructions can be implemented in just a couple lines of Python.

I spent a lot of time thinking about how to group the instructions for this discussion. Ultimately, I decided to order them numerically so that they appear in the same order here as they do in the code. Every instruction in CHIP-8 is 16 bits, or in other words, 2 bytes or 4 nibbles, so it translates to four hexadecimal digits. Any uppercase hexadecimal digit 0–F in an instruction is a literal. Any lowercase letter indicates a value that will be used as part of the implementation of the instruction. An underscore (_) indicates the nibble is arbitrary. The instructions were originally described in the RCA COSMAC VIP CDP18S711 Instruction Manual.[5]

NOTE *A few instructions listed here weren't present in the original CHIP-8 specification (for example, `8x_6` and `8x_E`). Their functionality sometimes differs across varying CHIP-8 implementations.*

Screen Clearing and Basic Jumps

The first set of instructions are used for cleaning up the entire screen all at once and for moving from one part of the program to another part of the program.

`00E0` Clear the screen.

`00EE` Return from a subroutine.

`0nnn` Call the program at `nnn`, reset the timers and registers, and clear the screen.

`1nnn` Jump to address `nnn` without resetting.

`2nnn` Call the subroutine at `nnn`.

Conditional Skips

The next set of instructions are for jumps to another part of the program if a particular condition is true.

`3xnn` Skip the next instruction if `v[x]` equals `nn`.

`4xnn` Skip the next instruction if `v[x]` doesn't equal `nn`.

`5xy_` Skip the next instruction if `v[x]` equals `v[y]`.

General-Purpose Register Adjustments, Arithmetic, and Bit Manipulation

Next come standard instructions that you would find in any CPU or VM for actions like doing math, setting registers, and shifting bits.

`6xnn` Set `v[x]` to `nn`.

`7xnn` Add `nn` to `v[x]`.

`8xy0` Set `v[x]` to `v[y]`.

8xy1 Set v[x] to v[x] | v[y] (bitwise OR).

8xy2 Set v[x] to v[x] & v[y] (bitwise AND).

8xy3 Set v[x] to v[x] ^ v[y] (bitwise XOR).

8xy4 Add v[y] to v[x] and set the carry flag.

8xy5 Subtract v[y] from v[x] and set the borrow flag.

8x_6 Shift v[x] right one bit and set the flag to the least-significant bit.

8xy7 Subtract v[x] from v[y] and store the result in v[x]; set the borrow flag.

8x_E Shift v[x] left one bit and set the flag to the most-significant bit.

Miscellaneous Instructions

These instructions don't quite have a unified subject area, but their opcodes are close to one another numerically.

9xy0 Skip the next instruction if v[x] doesn't equal v[y].

Annn Set i to nnn.

Bnnn Jump to nnn + v[0].

Cxnn Set v[x] to a random integer (0–255) & nn (bitwise AND).

Dxyn Draw a sprite that's n high at (v[x], v[y]); set the flag on a collision.

Key and Timer Instructions

The next batch of instructions are for manipulating the VM's timers and checking on the status of various keys or waiting for a particular key to be pressed.

Ex9E Skip the next instruction if key v[x] is set (pressed).

ExA1 Skip the next instruction if key v[x] is not set (not pressed).

Fx07 Set v[x] to the delay timer.

Fx0A Wait until the next key press, then store the key in v[x].

Fx15 Set the delay timer to v[x].

Fx18 Set the sound timer to v[x].

Register i Instructions

All the instructions in this last set are related to the memory index register (i).

Fx1E Add v[x] to i.

Fx29 Set i to the location of character v[x] in the font set.

Fx33 Store the binary-coded decimal (BCD) value in v[x] at memory locations i, i + 1, and i + 2. (See the "Binary-Coded Decimal" box on page 122 for more on this.)

Fx55 Dump registers v[0] through v[x] in memory, starting at i.

Fx65 Store memory from i through i + x in registers v[0] through v[x].

Consider for a moment how mundane these instructions sound. You really don't need any sophisticated mechanisms to have a working "computer." Contrast the 35 CHIP-8 instructions described here with the 8 instructions in our implementation of Brainfuck from Chapter 1. Both are memory-constrained Turing machines, and they aren't as different from each other as their superficial instruction syntax may make it appear.

BINARY-CODED DECIMAL

Binary-coded decimal (BCD) is a way of storing decimal numbers in binary. It's not widely used today, but it was common in early computers. For example, several microprocessors from the 1970s included explicit instructions for BCD arithmetic, which offered more precision when dealing with decimal rounding and to some extent made machine code more readable. For the average modern programmer, there isn't much value in learning BCD except as a curiosity. There were multiple different BCD schemes, and frankly I don't think that learning the particular scheme used in the CHIP-8 VM is a valuable use of our space in this book.

The Implementation

Now that we know the CHIP-8 architecture, we're ready to implement our VM. The file *__main__.py* will contain the main run loop that handles user input, updates the display, manages timers, and most importantly, tells the VM to step through the next instruction. This file is also where the command line argument that specifies the ROM file is parsed. Meanwhile, *vm.py* is the actual VM.

ROMS

Did you ever wonder why the files that hold games used in emulators are called ROMs? *ROM* stands for *read-only memory*. Most early video game systems used plastic cartridges that were glorified holders for ROM chips that directly plugged into the consoles. When the games were converted into files for emulators, someone would have to go and plug the ROM chip into a specialized device connected to their computer and "rip" the data from the ROM chip to store it in a file. The file would have an exact copy of the data on the ROM chip, perhaps with some extra header information depending on the emulation ecosystem.

While the original ROM chips couldn't have their data modified, these "ROM files" are just like any other files and can be modified to change the games. Hence, the subculture of *ROM hacking*, in which developers change the graphics or gameplay of games meant to be run in emulators.

We'll utilize two external libraries in our implementation. Pygame, a Python library designed for game development, provides an easy way to get a window on the screen, fill that window with the pixels from our VM's display, and handle keyboard input. NumPy, a numerical computing library, can help create the two-dimensional array used as the backing buffer for the Pygame window's pixels. This array will serve as the "graphics RAM" of our VM. Pygame natively works with NumPy arrays, and NumPy arrays are more performant than anything in the Python standard library for representing this buffer. Make sure you've installed Pygame and NumPy before running the program.

Like replicating a file format in Chapter 3, implementing a VM or emulator requires a fair amount of low-level bit manipulation. See the appendix to read up on Python's bitwise operators.

The Run Loop

The run loop is responsible for advancing the VM by one instruction, redrawing the screen, handling any events (key presses to be passed to the VM), playing the beep sound, and updating CHIP-8's two timers. Pygame makes drawing, playing sounds, and reading keyboard input almost trivial; it's a very easy-to-use library. Let's start with some initialization code and continue through to the beginning of the run loop:

Chip8/__main__.py

```
import sys
from argparse import ArgumentParser
from Chip8.vm import VM, SCREEN_WIDTH, SCREEN_HEIGHT
from Chip8.vm import TIMER_DELAY, FRAME_TIME_EXPECTED, ALLOWED_KEYS
import pygame
from timeit import default_timer as timer
import os

def run(program_data: bytes, name: str):
    # Startup Pygame, create the window, and load the sound
    pygame.init()
    screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT),
                                     pygame.SCALED)
    pygame.display.set_caption(f"Chip8 - {os.path.basename(name)}")
    bee_sound = pygame.mixer.Sound(os.path.dirname(os.path.realpath(__file__))
                                   + "/bee.wav")
    currently_playing_sound = False
    vm = VM(program_data) # load the virtual machine with the program data
```

```
timer_accumulator = 0.0 # used to limit the timer to 60 Hz
# Main virtual machine loop
while True:
    frame_start = timer()
    vm.step()
    if vm.needs_redraw:
        pygame.surfarray.blit_array(screen, vm.display_buffer)
        pygame.display.flip()
```

At the beginning of the run loop, the time is recorded with `frame_start = timer()` to measure the duration of each iteration of the loop. This is because CHIP-8's timers need to be decremented 60 times per second (if they're above zero). The VM is then told to execute an instruction (and therefore to move to the next instruction) via `vm.step()`. If indicated by `vm.needs_redraw`, the display is then redrawn via two simple calls to Pygame. One copies the VM's display buffer to the screen, and the other shows it.

Note that the code uses the term *frame* a little differently than is typical. In most programs, a frame is one full refresh of the entirety of the program's graphical output, but in this context, our run loop won't necessarily redraw the graphics every iteration, since `vm.needs_redraw` may not always be `True`.

What definitely *will* happen every "frame" is that one instruction will be executed as a result of the call to `vm.step()`. As such, I thought about using the word *instruction* rather than *frame* in this section of the code, for example, `instruction_start` rather than `frame_start`. However, more than just the execution of an instruction is happening in the run loop—there's also graphical output, keyboard handling, and sound output—so *instruction* sounded too limited. But again, *frame* isn't quite accurate either. It's true what they say: one of the hardest problems in computer science is naming.

The run loop finishes by handling keyboard events, playing a sound when the VM's Boolean `vm.play_sound` indicates, and handling timing:

```
    # Handle keyboard events
    for event in pygame.event.get():
        if event.type == pygame.KEYDOWN:
            key_name = pygame.key.name(event.key)
            if key_name in ALLOWED_KEYS:
                vm.keys[ALLOWED_KEYS.index(key_name)] = True
        elif event.type == pygame.KEYUP:
            key_name = pygame.key.name(event.key)
            if key_name in ALLOWED_KEYS:
                vm.keys[ALLOWED_KEYS.index(key_name)] = False
        elif event.type == pygame.QUIT:
            sys.exit()

    # Sound
    if vm.play_sound:
        if not currently_playing_sound:
            bee_sound.play(-1)
            currently_playing_sound = True
```

```
        else:
            currently_playing_sound = False
            bee_sound.stop()

        # Handle timing
        frame_end = timer()
        frame_time = frame_end - frame_start # time it took in seconds
        timer_accumulator += frame_time
        # Every 1/60 of a second decrement the timers
        if timer_accumulator > TIMER_DELAY:
          ❶ vm.decrement_timers()
            timer_accumulator = 0
        # Limit the speed of the entire machine to 500 "frames" per second
        if frame_time < FRAME_TIME_EXPECTED:
            difference = FRAME_TIME_EXPECTED - frame_time
          ❷ pygame.time.delay(int(difference * 1000))
            timer_accumulator += difference
```

Even though we aren't using frames to measure traditional frames per second (FPS), as you may be familiar with from gaming, the timing of each iteration is still important. We need to keep track of timing to ensure the VM's countdown timers are ticked every 1/60 of a second as required by the CHIP-8 specification ❶, and to limit the overall speed of the VM ❷. If the VM runs too fast, games will be unplayable since they were designed for the slow computers of the 1970s. You can adjust the speed of the VM, and therefore any software running on it, by changing the `FRAME_TIME_EXPECTED` constant in *vm.py*. In testing, I found that 500 "frames" per second, or in other words, each "frame" being approximately 1/500 of a second, to be a solid speed for most games.

Command Line Arguments

As in previous programs, we use `ArgumentParser` to handle command line arguments:

```
if __name__ == "__main__":
    # Parse the file argument
    file_parser = ArgumentParser("Chip8")
    file_parser.add_argument("rom_file",
                             help="A file containing a Chip-8 game.")
    arguments = file_parser.parse_args()
    with open(arguments.rom_file, "rb") as fp:
        file_data = fp.read()
        run(file_data, arguments.rom_file)
```

In this case, we have just a single command line argument—the name of the file containing the program data for the CHIP-8 VM. The file's raw bytes are read and passed to `run()`, where they in turn are passed to the constructor of the VM.

VM Setup and Helper Functions

We're ready for the actual VM implementation. We start, as we so often do, with some constants:

Chip8/vm.py

```
from array import array
from random import randint
import numpy as np
import pygame
import sys

RAM_SIZE = 4096  # in bytes, aka 4 kilobytes
SCREEN_WIDTH = 64
SCREEN_HEIGHT = 32
SPRITE_WIDTH = 8
WHITE = 0xFFFFFFFF
BLACK = 0
TIMER_DELAY = 1/60  # in seconds... about 60 Hz
FRAME_TIME_EXPECTED = 1/500  # for limiting VM speed
ALLOWED_KEYS = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                "a", "b", "c", "d", "e", "f"]

# The font set, hardcoded
FONT_SET = [
    0xF0, 0x90, 0x90, 0x90, 0xF0,  # 0
    0x20, 0x60, 0x20, 0x20, 0x70,  # 1
    0xF0, 0x10, 0xF0, 0x80, 0xF0,  # 2
    0xF0, 0x10, 0xF0, 0x10, 0xF0,  # 3
    0x90, 0x90, 0xF0, 0x10, 0x10,  # 4
    0xF0, 0x80, 0xF0, 0x10, 0xF0,  # 5
    0xF0, 0x80, 0xF0, 0x90, 0xF0,  # 6
    0xF0, 0x10, 0x20, 0x40, 0x40,  # 7
    0xF0, 0x90, 0xF0, 0x90, 0xF0,  # 8
    0xF0, 0x90, 0xF0, 0x10, 0xF0,  # 9
    0xF0, 0x90, 0xF0, 0x90, 0x90,  # A
    0xE0, 0x90, 0xE0, 0x90, 0xE0,  # B
    0xF0, 0x80, 0x80, 0x80, 0xF0,  # C
    0xE0, 0x90, 0x90, 0x90, 0xE0,  # D
    0xF0, 0x80, 0xF0, 0x80, 0xF0,  # E
    0xF0, 0x80, 0xF0, 0x80, 0x80   # F
]
```

Most of these constants are self-explanatory and in line with the original CHIP-8 specifications. The VM has 4KB of main memory. It specifies graphics in the form of a black-and-white output picture with a 64×32 resolution. The timers update 60 times per second. The original CHIP-8 systems had 16 keys you could press on the controller. We could probably arrange them in a more ergonomic way for gaming by mapping them to other keys, but in our implementation, we'll just leave the keys where they lie on the keyboard.

Probably the most unusual constant here is `FONT_SET`. This is 80 bytes of graphical data for displaying the digits 0–9 and the letters A–F. Each

character is specified by bits representing the pixels of the character should it be shown on the screen. Think of it as a primitive font that only has 16 characters. Several games expect this data to live in the first 80 bytes of memory so that they can write messages on the screen to the user.

Next, we have a helper function unrelated to the state of the VM:

```
def concat_nibbles(*args: int) -> int:
    result = 0
    for arg in args:
        result = (result << 4) | arg
    return result
```

The `concat_nibbles()` function takes an arbitrary number of integers and concatenates one after another by shifting each 4 bits to the left and bitwise OR-ing it with the next one. This will only be useful if the integers themselves are 4 bits. Suppose we have the integer `0111`. Shifting it 4 bits to the left will cause four zeros to follow the original 4 bits, as in `01110000`. Now suppose we have another 4-bit integer, `1010`. If we OR it with `01110000`, we obtain the result `01111010`, the concatenation of the original two 4-bit integers. We can keep doing this for an arbitrary number of 4-bit integers to concatenate them together.

Recall that a 4-bit integer is known as a *nibble*. The 16-bit instructions in CHIP-8 are divided into four nibbles, and each nibble often has a separate meaning. By default, we'll divide each instruction into its four constituent nibbles, but for a few instructions, we'll need to use the value of a few combined nibbles. Hence, the utility of the `concat_nibbles()` helper function.

The `VM` class starts with a constructor that initializes all of its mutable state including registers, RAM, the stack, the display buffer (what today we would call VRAM or video RAM), the timers, and a couple other helper variables:

```
class VM:
    def __init__(self, program_data: bytes):
        # Initialized registers and memory constructs
        # General Purpose Registers - CHIP-8 has 16 of these registers
        self.v = array('B', [0] * 16)
        # Index Register
        self.i = 0
        # Program Counter
        # Starts at 0x200 because addresses below that were
        # used for the VM itself in the original CHIP-8 machines
        self.pc = 0x200
        # Memory - the standard 4k on the original CHIP-8 machines
        self.ram = array('B', [0] * RAM_SIZE)
        # Load the font set into the first 80 bytes
        self.ram[0:len(FONT_SET)] = array('B', FONT_SET)
        # Copy program into RAM starting at byte 512 by convention
        self.ram[512:(512 + len(program_data))] = array('B', program_data)
        # Stack - in real hardware this is typically limited to
        # 12 or 16 PC addresses for jumps, but since we're on modern hardware,
        # ours can just be unlimited and expand/contract as needed
        self.stack = []
```

```
        # Graphics buffer for the screen - 64 x 32 pixels
        self.display_buffer = np.zeros((SCREEN_WIDTH, SCREEN_HEIGHT),
                                       dtype=np.uint32)
        self.needs_redraw = False
        # Timers - really simple registers that count down to 0 at 60 hertz
        self.delay_timer = 0
        self.sound_timer = 0
        # These hold the status of whether the keys are down
        # CHIP-8 has 16 keys
        self.keys = [False] * 16
```

A few of these state variables have important default values. For example, the program counter (`pc`) should always be set to location `0x200` (512 in decimal) since the first 512 bytes of memory in CHIP-8 machines were originally used for storing the CHIP-8 VM itself. This means CHIP-8 programs couldn't use that memory and had to start at byte 512. I've extensively commented the constructor to explain each variable as it's declared. Notice that the vast majority of our VM just uses the Python standard library for its implementation, except for `display_buffer`, which is a NumPy array. This is the format that Pygame expects.

Next, we have a trivial helper method, `decrement_timers()`, and a simple dynamic property, `play_sound`:

```
    def decrement_timers(self):
        if self.delay_timer > 0:
            self.delay_timer -= 1
        if self.sound_timer > 0:
            self.sound_timer -= 1

    @property
    def play_sound(self) -> bool:
        return self.sound_timer > 0
```

Both `decrement_timers()` and `play_sound` were used in the run loop we looked at earlier in *__main__.py*.

Graphics

CHIP-8 sees the screen as a 64×32 pixel plane with a cartesian coordinate system having the origin, location (0,0), in the top left, and the y-axis oriented downward. In other words, the x-coordinate increases as we travel from left to right and the y-coordinate increases as we travel from top to bottom. The bottom-right pixel is therefore at location (63,31). There are no negative coordinates, and it isn't possible to access pixel locations beyond the screen.

Each pixel is represented in memory as a single bit. In our implementation, a 1 represents a white pixel and a 0 represents a black pixel. The graphics memory (or "buffer") is separate from the main program memory and can only be manipulated indirectly using CHIP-8 instructions. Pygame uses 32-bit integers to represent pixels on the screen in RGBA format

(the *A* is for *alpha*, or transparency), so each of our 1-bit pixel values must become a 32-bit integer when we store it in the `display_buffer`.

CHIP-8 draws using *sprites*, which are little bitmaps (or images, if you like) that can move around the screen. Every sprite in CHIP-8 is 8 pixels wide and can be anywhere between 1 and 15 pixels high. Figure 5-1 illustrates an 8×3 sprite representing the word *HI* being drawn on the screen at location (28,15).

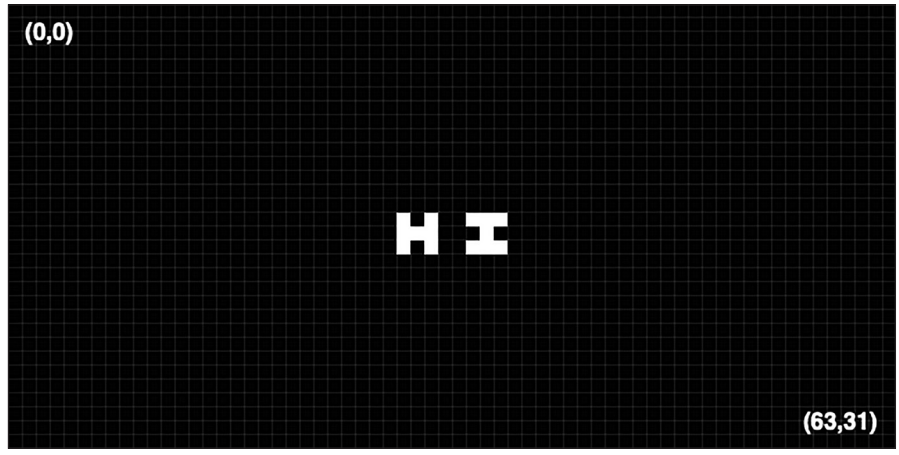

Figure 5-1: The word HI *as an 8×3 sprite*

Since each row in a CHIP-8 sprite is exactly 8 pixels, it's represented using 8 bits. Since 8 bits is 1 byte, each row of a sprite can therefore be represented by a single byte. Since the *HI* sprite is three rows high, it can be represented by 3 bytes. In binary, those 3 bytes would look like this:

```
10100111
11100010
10100111
```

Notice how each 1 maps to a white pixel and each 0 maps to a black pixel. With this information, hopefully the font set we defined earlier also makes more sense now: each character in the font set is just an 8×5 sprite.

Drawing sprites is the only way to modify the display buffer, other than clearing it, so the CHIP-8 VM has a single draw instruction, `Dxyn`. It draws a sprite of a specified height residing at the memory location specified by the `i` register. The `D` in the instruction is a constant nibble, and the `x` and `y` nibbles represent the indices into the `v` registers where the x- and y-coordinates for the top left of the sprite should be located. In other words, the x-coordinate is retrieved from register `v[x]` and the y-coordinate from register `v[y]`. The `n` nibble represents the height of the sprite. This is why sprites can't be taller than 15 pixels: a nibble is 4 bits, and 4 bits can maximally represent the number 15.

The nibbles of `Dxyn` correspond to the parameters of the `draw_sprite()` helper method:

```
# Draw a sprite at *x*, *y* using data at *i* with a height of *height*
def draw_sprite(self, x: int, y: int, height: int):
    flipped_black = False  # did drawing this flip any pixels?
    for row in range(0, height):
        row_bits = self.ram[self.i + row]
        for col in range(0, SPRITE_WIDTH):
            px = x + col
            py = y + row
            if px >= SCREEN_WIDTH or py >= SCREEN_HEIGHT:
                continue  # ignore off-screen pixels
            new_bit = (row_bits >> (7 - col)) & 1
            old_bit = self.display_buffer[px, py] & 1
            if new_bit & old_bit:  # if both set, flip white -> black
                flipped_black = True
            # CHIP-8 draws by XORing
            new_pixel = new_bit ^ old_bit
            self.display_buffer[px, py] = WHITE if new_pixel else BLACK
    # Set flipped flag for collision detection
    self.v[0xF] = 1 if flipped_black else 0
```

CHIP-8 draws sprites using XOR operations. *XOR*, or *exclusive or*, is a bitwise operation that returns a 1 if two bits are different and a 0 if they're the same. Python uses the ^ operator for XOR. Table 5-2 shows a truth table for XOR.

Table 5-2: XOR Truth Table

0 ^ 0	0 ^ 1	1 ^ 0	1 ^ 1
0	1	1	0

The CHIP-8 draw instruction takes a sprite and XORs its pixels with the pixels already on the screen at the location specified. If this screen location is all black pixels, this will effectively just draw the sprite. However, if the screen location contains some white pixels (1s), black pixels will be drawn where the white pixels of the sprite overlap with the white pixels of the screen. This is because 1 XOR 1 is 0. The CHIP-8 draw instruction tracks whether any of these overlaps occur (a screen white pixel was turned to a black pixel by drawing the sprite). If they do, it sets the flag register (`v[0xF]`).

The `draw_sprite()` method is a codification of this process. We iterate through all of the rows and columns of a sprite that begins at the memory location specified by register `i`, pulling out each pixel of the sprite using a right shift operation and storing it in `new_bit`. The `&` operation on the data going into `new_bit` ensures that only the single last bit of the shift operation is stored in `new_bit`. We compare each `new_bit` to the bit already on the screen, `old_bit`, and if an `old_bit` will be flipped from white to black, we set the flag register. We change the display buffer by taking the XOR of `new_bit` and `old_bit`.

Why do we need a flag to track whether drawing a sprite causes a previously lit screen pixel to be turned off? It's effectively a form of collision detection. If a sprite hits something that was already on the screen, that's particularly helpful to know in a game. For example, if you are programming a tennis game, you would want to know when the ball moves and hits a racket already on the screen.

Instruction Execution

Now it's time for the heart of the VM. We have one method left, but it's a big one: we need to implement all of the VM's instructions. This isn't dissimilar to executing the statements in our interpreters in Chapters 1 and 2. Whether executing interpreter statements, VM instructions, or microprocessor opcodes in an emulator, we need to do something pretty simple: recognize what the next instruction is and then execute a different few lines of code that manipulate the state of the VM based on its intended operation.

For example, if we see an add instruction, we should add the two specified numbers together and store the result in a specified location. If we see a jump instruction, we should move execution to a specified location in memory. It's literally about recognizing what instruction is being executed and changing a few state variables representing memory, registers, and the like based on that instruction. The simplest way to do this would be with a large number of `if` statements. The pseudocode may look like this:

```
if instruction == ADD:
    add some numbers together and store the sum
elif instruction == JUMP:
    jump to a location by changing the program counter
elif instruction == DRAW:
    draw the sprite where specified by changing the display buffer
etc.
```

Beyond using a bunch of `if` statements, there are three common patterns for writing the code that executes the instructions. The first is a giant `switch` statement, a construct present in many languages but not quite in Python in the same form. I assume most readers have seen a `switch` statement before in a language like C or Java. If you haven't, you can think of it as a primitive form of Python's `match` statement like we used in Chapters 1 and 2. The case of the `switch` statement that executes is dependent on the instruction. This is somewhat similar to the pseudocode just shown. In fact, prior to the introduction of the `match` statement in Python 3.10, the way you would implement this pattern in Python was indeed with a ton of `if` and `elif` clauses. This is the simplest way to implement instruction execution, but it can become unwieldy for a large instruction set.

The next pattern is to use a *jump table*, which consists of an array of function pointers. We index into the array depending on the instruction and then execute the appropriate function that's returned. Instructions are just integers, which is why they can be used as array indices. If the instructions were strings for some reason, we could instead use a dictionary where

the keys are instructions and the values are function pointers, although this is a bit less efficient. Because this pattern divides the work across many helper functions, it generally results in cleaner code than a giant `switch` statement and may be preferred for a larger instruction set.

The third pattern is to use *dynamic recompilation*, where we translate each instruction into an instruction that the underlying hardware understands (or something that can further be translated into such). For example, if we have an addition instruction in the VM running on an x86 microprocessor, we may translate the VM's addition instruction into the machine code for an equivalent x86 addition instruction. This is the most complicated pattern to implement because it requires intimate knowledge of not just the original instruction set but also the instruction set being translated into. It will, however, result in the fastest performance.

In this program, we'll use a giant `match` statement since CHIP-8's instruction set is relatively small. When we create an NES emulator in the next chapter, we'll use a jump table because the 6502 microprocessor has an instruction set that's roughly double the size (although still much smaller than almost any other microprocessor). Dynamic recompilation is a significantly more complicated technique and beyond the scope of this book.

The `step()` method is responsible for executing instructions, but first the method needs to retrieve the next instruction to execute:

```
    def step(self):
        # We look at the opcode in terms of its nibbles (4 bit pieces)
        # Opcode is 16 bits made up of next two bytes in memory
        first2 = self.ram[self.pc]
        last2 = self.ram[self.pc + 1]
        first = (first2 & 0xF0) >> 4
        second = first2 & 0xF
        third = (last2 & 0xF0) >> 4
        fourth = last2 & 0xF

        self.needs_redraw = False
        jumped = False
```

The next instruction is located at the memory address stored in the program counter (`pc`). Since instructions consist of 16 bits, we retrieve the next 2 bytes at `pc` and store them in `first2` and `last2`. As discussed earlier, it's convenient to think about each CHIP-8 instruction as a combination of four nibbles, since each individual nibble is meaningful for many of the instructions. We store the nibbles in `first`, `second`, `third`, and `fourth`. All of the pattern-matching around our instructions will be in terms of nibbles.

As we execute the instruction, we'll also be keeping track of whether it requires any redrawing through `needs_redraw` and whether it modified `pc` through `jumped`. The run loop uses `needs_redraw` as an optimization. Why do any drawing when nothing changed? Keeping track of `jumped` allows for some common code to be at the bottom of `step()`, reducing a little bit of code duplication.

Now we arrive at the actual instructions. The giant `match` statement is upon us. Our implementation utilizes Python's elegant `match` syntax to capture

the nibbles that are necessary for the execution of an instruction in temporary variables. The details of each instruction's execution follow directly from its description earlier in the chapter. Many of the instructions are able to be implemented in just a single line of code. It would be exceedingly dry to write about each of them in turn. Instead, what follows is a reproduction of the rest of `step()`, with comments providing a bit of additional context.

Before you look at the code, though, this is a good place to stop and try to implement the instructions yourself. You don't have to use a `match` statement. You could use a series of `if...elif` statements as I did in Python 3.9 before the `match` statement existed. (I tested and there was virtually no performance difference between the two.) You already have all the setup you need to be able to concentrate only on what each instruction is supposed to do instead of configuring the system's memory or register representation. You don't need to think about loading the ROM file or what some constants should be. Just think about logic and how each operation would modify the VM's state.

Some of the descriptions of the instructions earlier in this chapter were fairly brief, but you can find more detailed instructions in any of a myriad of CHIP-8 references online. Don't spend too much time on a single instruction, though. You can always look at the implementation here if you get stuck. After you try writing your own instruction implementations, you can return to this book's code to double-check your work. Doing this work yourself first will give you a good idea of what goes into writing a simple VM or emulator. Don't be afraid: you'll be amazed at how simple it is to implement many of the instructions. Remember, the original CHIP-8 VM fit in just 512 bytes of memory!

```
        match (first, second, third, fourth):
            case (0x0, 0x0, 0xE, 0x0):  # display clear
                self.display_buffer.fill(0)
                self.needs_redraw = True
            case (0x0, 0x0, 0xE, 0xE):  # return from subroutine
                self.pc = self.stack.pop()
                jumped = True
            case (0x0, n1, n2, n3):  # call program
                self.pc = concat_nibbles(n1, n2, n3)  # go to start
                # Clear registers
                self.delay_timer = 0
                self.sound_timer = 0
                self.v = array('B', [0] * 16)
                self.i = 0
                # Clear screen
                self.display_buffer.fill(0)
                self.needs_redraw = True
                jumped = True
            case (0x1, n1, n2, n3):  # jump to address
                self.pc = concat_nibbles(n1, n2, n3)
                jumped = True
            case (0x2, n1, n2, n3):  # call subroutine
                self.stack.append(self.pc + 2)  # put return place on stack
                self.pc = concat_nibbles(n1, n2, n3)  # goto subroutine
                jumped = True
```

```
            case (0x3, x, _, _):  # conditional skip v[x] equal last2
                if self.v[x] == last2:
                    self.pc += 4
                    jumped = True
            case (0x4, x, _, _):  # conditional skip v[x] not equal last2
                if self.v[x] != last2:
                    self.pc += 4
                    jumped = True
            case (0x5, x, y, _):  # conditional skip v[x] equal v[y]
                if self.v[x] == self.v[y]:
                    self.pc += 4
                    jumped = True
            case (0x6, x, _, _):  # set v[x] to last2
                self.v[x] = last2
            case (0x7, x, _, _):  # add last2 to v[x]
                self.v[x] = (self.v[x] + last2) % 256
            case (0x8, x, y, 0x0):  # set v[x] to v[y]
                self.v[x] = self.v[y]
            case (0x8, x, y, 0x1):  # set v[x] to v[x] | v[y]
                self.v[x] |= self.v[y]
            case (0x8, x, y, 0x2):  # set v[x] to v[x] & v[y]
                self.v[x] &= self.v[y]
            case (0x8, x, y, 0x3):  # set v[x] to v[x] ^ v[y]
                self.v[x] ^= self.v[y]
            case (0x8, x, y, 0x4):  # add with carry flag
                try:
                    self.v[x] += self.v[y]
                    self.v[0xF] = 0  # indicate no carry flag
                except OverflowError:
                    self.v[x] = (self.v[x] + self.v[y]) % 256
                    self.v[0xF] = 1  # set carry flag
            case (0x8, x, y, 0x5):  # subtract with borrow flag
                try:
                    self.v[x] -= self.v[y]
                    self.v[0xF] = 1  # indicate no borrow (yes, weird it's 1)
                except OverflowError:
                    self.v[x] = (self.v[x] - self.v[y]) % 256
                    self.v[0xF] = 0  # indicates there was a borrow
            case (0x8, x, _, 0x6):  # v[x] >> 1 v[f] = least significant bit
                self.v[0xF] = self.v[x] & 0x1
                self.v[x] >>= 1
            case (0x8, x, y, 0x7):  # subtract with borrow flag (y - x in x)
                try:
                    self.v[x] = self.v[y] - self.v[x]
                    self.v[0xF] = 1  # indicate no borrow (yes, weird it's 1)
                except OverflowError:
                    self.v[x] = (self.v[y] - self.v[x]) % 256
                    self.v[0xF] = 0  # indicates there was a borrow
            case (0x8, x, _, 0xE):  # v[x] << 1 v[f] = most significant bit
                self.v[0xF] = (self.v[x] & 0b10000000) >> 7
                self.v[x] = (self.v[x] << 1) & 0xFF
            case (0x9, x, y, 0x0):  # conditional skip if v[x] != v[y]
                if self.v[x] != self.v[y]:
                    self.pc += 4
                    jumped = True
```

```
            case (0xA, n1, n2, n3):  # set i to address n1n2n3
                self.i = concat_nibbles(n1, n2, n3)
            case (0xB, n1, n2, n3):  # jump to n1n2n3 + v[0]
                self.pc = concat_nibbles(n1, n2, n3) + self.v[0]
                jumped = True
            case (0xC, x, _, _):  # v[x] = random number (0-255) & last2
                self.v[x] = last2 & randint(0, 255)
            case (0xD, x, y, n):  # draw sprite at (vx, vy) that's n high
                self.draw_sprite(self.v[x], self.v[y], n)
                self.needs_redraw = True
            case (0xE, x, 0x9, 0xE):  # conditional skip if keys(v[x])
                if self.keys[self.v[x]]:
                    self.pc += 4
                    jumped = True
            case (0xE, x, 0xA, 0x1):  # conditional skip if not keys(v[x])
                if not self.keys[self.v[x]]:
                    self.pc += 4
                    jumped = True
            case (0xF, x, 0x0, 0x7):  # set v[x] to delay_timer
                self.v[x] = self.delay_timer
            case (0xF, x, 0x0, 0xA):  # wait until next key then store in v[x]
                # Wait here for the next key then continue
                while True:
                    event = pygame.event.wait()
                    if event.type == pygame.QUIT:
                        sys.exit()
                    if event.type == pygame.KEYDOWN:
                        key_name = pygame.key.name(event.key)
                        if key_name in ALLOWED_KEYS:
                            self.v[x] = ALLOWED_KEYS.index(key_name)
                            break
            case (0xF, x, 0x1, 0x5):  # set delay_timer to v[x]
                self.delay_timer = self.v[x]
            case (0xF, x, 0x1, 0x8):  # set sound_timer to v[x]
                self.sound_timer = self.v[x]
            case (0xF, x, 0x1, 0xE):  # add vx to i
                self.i += self.v[x]
            case (0xF, x, 0x2, 0x9):  # set i to location of character v[x]
                self.i = self.v[x] * 5  # built-in font set is 5 bytes apart
            case (0xF, x, 0x3, 0x3):  # store BCD at v[x] in i, i+1, i+2
                self.ram[self.i] = self.v[x] // 100  # 100s digit
                self.ram[self.i + 1] = (self.v[x] % 100) // 10  # 10s digit
                self.ram[self.i + 2] = (self.v[x] % 100) % 10  # 1s digit
            case (0xF, x, 0x5, 0x5):  # reg dump v0 to vx starting at i
                for r in range(0, x + 1):
                    self.ram[self.i + r] = self.v[r]
            case (0xF, x, 0x6, 0x5):  # store i through i+r in v0 through vr
                for r in range(0, x + 1):
                    self.v[r] = self.ram[self.i + r]
            case _:
                print(f"Unknown opcode {(hex(first), hex(second),
                                         hex(third), hex(fourth))}!")

        if not jumped:
            self.pc += 2  # increment program counter
```

At the end of `step()`, we increment the program counter if we didn't jump. This ensures that we'll have moved on to the next instruction the next time `step()` is called. Since each CHIP-8 instruction is 2 bytes long, the program counter is incremented by `2`. If there was a jump, then execution was directly moved to a specific different instruction somewhere else in memory.

Testing the VM

The most granular way to test the VM would be to write our own unit tests for each of the instructions. For each test, we would try running an instruction and then verify that the subsequent internal state of the VM was correct. While this would be ideal, in the interests of time and space we'll instead do something more akin to integration tests: we'll see how our VM performs running real CHIP-8 programs. Do they run correctly?

As it happens, there are even test ROMs that offer a kind of one-stop shop for testing a CHIP-8 VM. Two such test ROMs are included in the *Chip8/Tests* subdirectory of the book's source code repository. Both test ROMs were released under open licenses by their developers, and those licenses are included in the subdirectories. Let's run the first test ROM from the repository's home directory:

```
% python3 -m Chip8 Chip8/Tests/chip8-test-rom/test_opcode.ch8
```

If the VM is working correctly, you should see a screen of OKs, as shown in Figure 5-2.

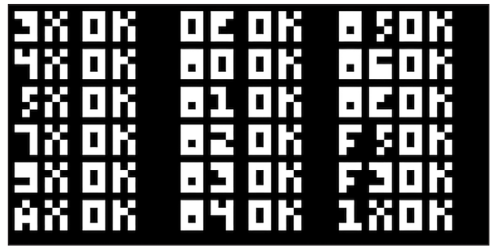

Figure 5-2: Running the first test ROM

Now let's check our work with the second test ROM:

```
% python3 -m Chip8 Chip8/Tests/chip8-test-rom-2/chip8-test-rom.ch8
```

This one just displays OK a single time in the upper-left corner (see Figure 5-3).

Figure 5-3: Running the second test ROM

These tests aren't comprehensive, but they're a good starting point. Now it's time for the ultimate integration tests: Can our VM accurately play games?

Playing Games

The *Chip8/Games* subdirectory of the book's repository contains a selection of CHIP-8 ROMs that have been placed into the public domain. If you find the control schemes of some of them a bit unwieldy, consider changing the default key bindings. Right now, `ALLOWED_KEYS` are read directly from their respective keys, so an *A* in the VM is the *A* key on the keyboard. The systems these were played on could have quite different key layouts, though, so a different scheme might be better for some of the games.

Most of the games are quite simple, which makes sense given the constraints of the hardware the VM was originally meant to run on. There are clones of popular games for more capable systems. First we have *BLINKY*, a kind of *Pac-Man* clone (Figure 5-4).

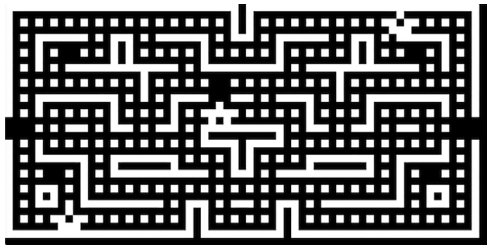

Figure 5-4: The BLINKY *game running on the VM*

INVADERS is a clone of *Space Invaders* (Figure 5-5).

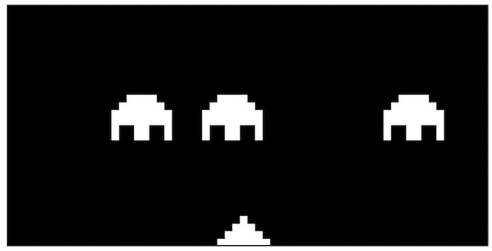

Figure 5-5: The INVADERS *game running on the VM*

VBRIX is a vertical form of *Breakout* (Figure 5-6).

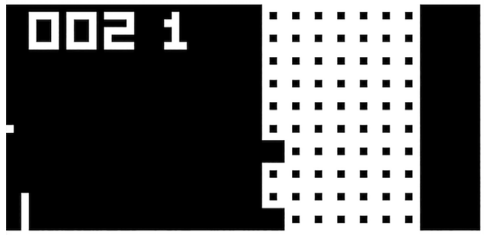

Figure 5-6: The VBRIX *game running on the VM*

And then there's *PONG* (Figure 5-7).

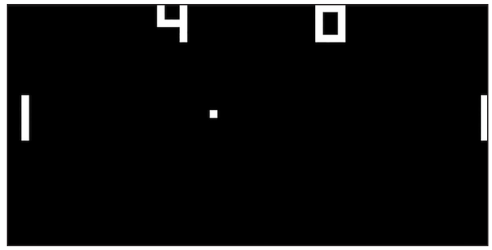

Figure 5-7: The PONG *game running on the VM*

There are several more games for you to check out bundled with the source code repository. Note the file sizes: most of these games are 500 bytes or less! The largest, *BLINKY*, is just 2KB.

CODE MEETS LIFE

I was always interested in developing my own emulator, but I didn't feel confident enough to build one until well into my programming life. When I started researching how to write an emulator, the standard advice I found was to first try writing a CHIP-8 VM since doing so is easier than writing almost any emulator but requires all the same elements (handling opcodes, simulating memory and registers, graphics, and so on).

I found an online tutorial that was reasonably good. I decided that I wanted to make it a little more challenging, though, so I developed my initial CHIP-8 VM in the then-new language Swift, which I was doing a lot of my professional work in at the time. It was a weekend project, the launching point that I needed to get started developing emulators.

Real-World Applications

VMs are ubiquitous in both historical and modern software development. Their chief advantage is portability. A program written for a VM will run on any platform that has an implementation of that VM. VMs also provide infrastructure that reduces the burden on a language author by eliminating the need to implement common language runtime features like garbage collection.

An early example was the compilation of Pascal by some compilers in the 1970s and 1980s to so-called *p-code* (a type of bytecode) that would run on a p-code VM. Two prominent modern VM environments are the JVM, mentioned earlier in this chapter, and Microsoft's competing Common Language Runtime (CLR), which is part of its .NET platform. Both the JVM and CLR are targeted by multiple popular programming languages. For example, C#, F#, and Visual Basic are languages that commonly target the CLR, but there are also implementations of popular languages like Python and Swift for the CLR.

Why do these language implementations compile into bytecode for the CLR instead of machine code? Once compiled, that bytecode can run on any platform that has an installed CLR. That's a kind of instant portability post-compilation. In addition, a sophisticated VM like the CLR will provide

language services like garbage collection, multithreading, and security mechanisms. Finally, when a VM like the CLR just-in-time (JIT) compiles intermediate code into machine code, it will apply optimizations that the language author doesn't need to think about.

Beyond abstract machines utilized as language runtimes, the term *virtual machine* is also confusingly used to refer to a whole hardware implementation in software—in other words, an emulator. Building an emulator is the subject of the next chapter.

Exercises

1. Try measuring the performance of the main opcode interpreter code using three different methodologies: the already implemented `match` statement, a series of `if...elif` statements, and a jump table. Determine which method is fastest using either a profiler or a simple timer. You may need to turn off the timing code in the main run loop in order to do this, or you may do this using a set of unit tests.
2. There's a slightly extended version of CHIP-8, known as SCHIP (SuperChip). It requires implementing a few more opcodes and changing a few elements of the original CHIP-8 VM, such as its resolution. Look up documentation for SCHIP and try turning our CHIP-8 VM into an SCHIP VM. Then, try playing some SCHIP games!
3. Try writing a very simple game that just displays a couple letters on the screen using CHIP-8's machine code instructions. You'll need a hex editor to do this. It's gratifying to see binary code you wrote running in a VM you understand.

Notes

1. Joe Weisbecker, "A Practical, Low-Cost, Home/School Microprocessor System," *Computer* 7, no. 08 (August 1974): 20–31.
2. Katianne Williams, "Joyce Weisbecker: The First Indie Game Developer," *IEEE Women in Engineering Magazine* 16, no. 2 (December 2022): 15–20, doi:10.1109/MWIE.2022.3203181.
3. RCA COSMAC VIP CDP18S711 Instruction Manual (RCA Corporation, 1978).
4. RCA COSMAC VIP CDP18S711 Instruction Manual (RCA Corporation, 1978).
5. RCA COSMAC VIP CDP18S711 Instruction Manual (RCA Corporation, 1978).